SMT-Based Constraint Answer Set Solver EZSMT (System Description)

Benjamin Susman[1] and Yuliya Lierler[2]

1 **University of Nebraska at Omaha, Dept. of Computer Science, Omaha, USA**
bensusman@gmail.com
2 **University of Nebraska at Omaha, Dept. of Computer Science, Omaha, USA**
ylierler@unomaha.edu

Abstract

Constraint answer set programming is a promising research direction that integrates answer set programming with constraint processing. Recently, the formal link between this research area and satisfiability modulo theories (or SMT) was established. This link allows the cross-fertilization between traditionally different solving technologies. The paper presents the system EZSMT, one of the first SMT-based solvers for constraint answer set programming. It also presents the comparative analysis of the performance of EZSMT in relation to its peers including solvers EZCSP, CLINGCON, and MINGO. Experimental results demonstrate that SMT is a viable technology for constraint answer set programming.

1998 ACM Subject Classification D.1.6 [Programming Techniques] Logic Programming, D.3.2 [Programming Languages] Language Classifications – Constraint and Logic Languages, F.4.1 [Mathematical Logic and Formal Languages] Mathematical Logic – Logic and Constraint Programming

Keywords and phrases constraint answer set programming, constraint satisfaction processing, satisfiability modulo theories

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.1

1 Introduction

Constraint answer set programming (CASP) is an answer set programming paradigm that integrates traditional answer set solving techniques with constraint processing. In some cases this approach yields substantial performance gains. Such systems as CLINGCON [11], EZCSP [1], and INCA [7] are fine representatives of CASP technology. Satisfiability Modulo Theories (SMT) is a related paradigm that integrates traditional satisfiability solving techniques with specialized "theory solvers" [4]. It was shown that under a certain class of SMT theories, these areas are closely related [14].

Lierler and Susman [14] presented theoretical grounds for using SMT systems for computing answer sets of a broad class of "tight" constraint answer set programs. Here we develop a system EZSMT that roots on that theoretical basis. The EZSMT solver takes as an input a constraint answer set program and translates it into so called constraint formula using the method related to forming logic program's completion. Lierler and Susman illustrated that constraint formulas coincide with formulas that SMT systems process. To interface with the SMT solvers, EZSMT utilizes the standard SMT-LIB language [4]. The empirical results carried out within this project suggest that SMT technology forms a powerful tool for finding solutions to programs expressed in CASP formalism. In particular, we compare the performance of EZSMT with such systems as EZCSP, CLINGCON, MINGO [16], and CMODELS [12]

Table 1 Example definitions for signature, lexicon, valuation.

Σ_1	$\{s, r, E, Q\}$
D_1	$\{1, 2, 3\}$
ρ_1	a function that maps $\Sigma_{1\|r}$ into relations $E^{\rho_1} = \{\langle 1\rangle\}$, $Q^{\rho_1} = \{\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle\}$
$\mathcal{L}_1$	lexicon (Σ_1, D_1, ρ_1)
ν_1	valuation over $\mathcal{L}_1$, where $s^{\nu_1} = 1$ and $r^{\nu_1} = 1$
ν_2	valuation over $\mathcal{L}_1$, where $s^{\nu_2} = 2$ and $r^{\nu_2} = 1$

on six benchmarks that have been previously shown to be challenging for "pure" answer set programming approaches (exhibited by answer set solver CMODELS in experiments of this paper). The experimental analysis of this work compares and contrasts three distinct areas of automated reasoning, namely, (constraint) answer set programming, SMT, and integer mixed programming. This can be seen as one more distinct contribution of this work.

The outline of the paper follows. We review the concepts of generalized constraint satisfaction problems, logic programs, and input completion. We next introduce so called EZ constraint answer set programs and EZ constraint formulas. We subsequently present the EZSMT solver and conclude by discussing empirical results comparing EZSMT to other leading CASP systems.

2 Preliminaries

Generalized Constraint Satisfaction Problems. We now present primitive constraints as defined by Marriott and Stuckey [17] using the notation convenient for our purposes. We refer to this concept as a constraint dropping the word "primitive". We then define a generalized constraint satisfaction problem that Marriott and Stuckey refer to as "constraint".

Signature, lexicon, constraints. We adopt the following convention: for a function ν and an element x, by x^ν we denote the value that function ν maps x to (in other words, $x^\nu = \nu(x)$).

A *domain* is a *nonempty* set of elements (values). A *signature* Σ is a set of *variables*, *predicate symbols*, and *function symbols (or f-symbols)*. Predicate and function symbols are associated with a positive integer called *arity*. By $\Sigma_{|v}$, $\Sigma_{|r}$, and $\Sigma_{|f}$ we denote the subsets of Σ that contain all variables, all predicate symbols, and all f-symbols respectively. For instance, Table 1 includes the definition of sample signature Σ_1, where s and r are variables, E is a predicate symbol of arity 1, and Q is a predicate symbol of arity 2. Then, $\Sigma_{1|v} = \{s, r\}$, $\Sigma_{1|r} = \{E, Q\}$, $\Sigma_{1|f} = \emptyset$.

A *lexicon* is a tuple (Σ, D, ρ, ϕ), where (i) Σ is a signature, (ii) D is a domain, (iii) ρ is a function from $\Sigma_{|r}$ to relations on D so that an n-ary predicate symbol is mapped into an n-ary relation on D, and (iv) ϕ is a function from $\Sigma_{|f}$ to functions so that an n-ary f-symbol is mapped into a function $D^n \to D$. For a lexicon $\mathcal{L} = (\Sigma, D, \rho, \phi)$, we call any function $\nu : \Sigma_{|v} \to D$ a *valuation over* $\mathcal{L}$. Table 1 presents definitions of sample domain D_1, function ρ_1, lexicon $\mathcal{L}_1$, and valuations ν_1 and ν_2 over $\mathcal{L}_1$. A *term* over signature Σ and domain D (or a lexicon (Σ, D, ρ, ϕ)) is either (i) a variable in $\Sigma_{|v}$, (ii) a domain element in D, or (iii) an expression $f(t_1, \ldots, t_n)$, where f is an f-symbol of arity n in $\Sigma_{|f}$ and $t_1, \ldots, t_n$ are terms over $[\Sigma, D]$.

A *constraint* is defined over lexicon $\mathcal{L} = (\Sigma, D, \rho, \phi)$. *Syntactically*, a constraint is an

expression of the form

$$P(t_1, \ldots, t_n), \text{ or} \tag{1}$$
$$\neg P(t_1, \ldots, t_n), \tag{2}$$

where P is a predicate symbol from $\Sigma_{|r}$ of arity n and $t_1, \ldots, t_n$ are terms over $\mathcal{L}$. *Semantically*, we first specify recursively a value that valuation ν over lexicon (Σ, D, ρ, ϕ) assigns to a term τ over the same lexicon. We denote this value by $\tau^{\nu,\phi}$ and define it as follows:

- for a term that is a variable x in $\Sigma_{|v}$, $x^{\nu,\phi} = x^{\nu}$,
- for a term that is a domain element d in D, $d^{\nu,\phi}$ is d itself,
- for a term τ of the form $f(t_1, \ldots, t_n)$, $f(t_1, \ldots, t_n)^{\nu,\phi} = f^{\phi}(t_1^{\nu,\phi}, \ldots, t_n^{\nu,\phi})$.

Second, we define what it means for valuation to be a solution of a constraint over the same lexicon. We say that ν over lexicon $\mathcal{L}$ *satisfies (is a solution to)* constraint of the form (1) over $\mathcal{L}$ when $\langle t_1^{\nu,\phi}, \ldots, t_n^{\nu,\phi}\rangle \in P^{\rho}$. Valuation ν *satisfies* constraint of the form (2) when $\langle t_1^{\nu,\phi}, \ldots, t_n^{\nu,\phi}\rangle \notin P^{\rho}$.

For instance, expressions

$$\neg E(s), \quad \neg E(2), \quad Q(r, s) \tag{3}$$

are constraints over lexicon $\mathcal{L}_1$. Valuation ν_1 satisfies constraints $\neg E(2)$, $Q(r, s)$, but does not satisfy $\neg E(s)$. Valuation ν_2 satisfies constraints $\neg E(s)$, $\neg E(2)$, but does not satisfy $Q(r, s)$.

A *generalized constraint satisfaction problem (GCSP)* $\mathcal{C}$ is a finite set of constraints over the same lexicon. By $\mathcal{L}^{\mathcal{C}}$ we denote the lexicon of constraints in GCSP $\mathcal{C}$. We say that a valuation ν over $\mathcal{L}^{\mathcal{C}}$ *satisfies (is a solution to)* GCSP $\mathcal{C}$, when ν is a solution to every constraint in $\mathcal{C}$. For example, any subset of constraints in (3) forms a GCSP. Sample valuation ν_2 from Table 1 satisfies GCSP composed of the first two constraints in (3). Neither ν_1 nor ν_2 satisfies the GCSP composed of all of the constraints in (3).

It is worth noting that syntactically, constraints are similar to *ground* literals of SMT. (In predicate logic, variables as defined here are referred to as *object constants* or *function symbols of 0 arity*.) Lierler and Susman [14] illustrated that given a GCSP $\mathcal{C}$ one can construct the "uniform" Σ-theory $U(\mathcal{C})$ based on the last three elements the GCSP's lexicon. Semantically, a GCSP $\mathcal{C}$ can be understood as a conjunction of its elements so that $U(\mathcal{C})$-models (as understood in SMT) of this conjunction coincide with solutions of $\mathcal{C}$.

Linear and Integer Linear Constraints. We now define "numeric" signatures and lexicons and introduce linear constraints that are commonly used in practice. The EZSMT system provides support for such constraints.

A *numeric signature* is a signature that satisfies the following requirements (i) its only predicate symbols are $<, >, \leq, \geq, =, \neq$ of arity 2, and (ii) its only f-symbols are $+, \times$ of arity 2. We use the symbol $\mathcal{A}$ to denote a numeric signature. Let $\mathbb{Z}$ and $\mathbb{R}$ be the sets of integers and real numbers respectively. Let $\rho_{\mathbb{Z}}$ and $\rho_{\mathbb{R}}$ be functions that map predicate symbols $\{<, >, \leq, \geq, =, \neq\}$ into usual arithmetic relations over integers and over reals, respectively. Let $\phi_{\mathbb{Z}}$ and $\phi_{\mathbb{R}}$ be functions that map f-symbols $\{+, \times\}$ into usual arithmetic functions over integers and over reals, respectively. We can now define the following lexicons (i) an *integer lexicon* of the form $(\mathcal{A}, \mathbb{Z}, \rho_{\mathbb{Z}}, \phi_{\mathbb{Z}})$, and a *numeric lexicon* of the form $(\mathcal{A}, \mathbb{R}, \rho_{\mathbb{R}}, \phi_{\mathbb{R}})$.

A *(numeric) linear expression* has the form

$$a_1 x_1 + \cdots + a_n x_n, \tag{4}$$

where $a_1, \ldots, a_n$ are real numbers and $x_1, \ldots, x_n$ are variables over real numbers. When $a_i = 1$ $(1 \leq i \leq n)$ we may omit it from the expression. We view expression (4) as an abbreviation for the following term $+(\times(a_1, x_1), +(\times(a_2, x_2), \cdots + (\times(a_{n-1}, x_{n-1}), \times(a_n, x_n))\ldots)$ over some numeric lexicon $(\mathcal{A}, \mathbb{R}, \rho_{\mathbb{R}}, \phi_{\mathbb{R}})$, where $\mathcal{A}$ contains $x_1, \ldots, x_n$ as its variables. For instance, $2x + 3y$ is an abbreviation for the expression $+(\times(2, x), \times(3, y))$. An *integer linear expression* has the form (4), where $a_1, \ldots, a_n$ are integers, and $x_1, \ldots, x_n$ are variables over integers.

We call a constraint *linear (integer linear)* when it is defined over some numeric (integer) lexicon and has the form $\bowtie (e, k)$ where e is a linear (integer linear) expression, k is a real number (an integer), and $\bowtie$ belongs to $\{<, >, \leq, \geq, =, \neq\}$. We can denote $\bowtie (e, k)$ using usual infix notation as follows $e \bowtie k$.

We call a GCSP a *(integer) linear constraint satisfaction problem* when it is composed of (integer) linear constraints. For instance, consider integer linear constraint satisfaction problem composed of two constraints $x > 4$ and $x < 5$ (here signature $\mathcal{A}$ is implicitly defined by restricting its variable to contain x). When the lexicon of GCSP is clear from the context, as in this example, we omit an explicit reference to it. It is easy to see that this problem has no solutions. On the other hand, linear constraint satisfaction problem composed of the same two constraints has infinite number of solutions, including valuation that assigns x to 4.1.

Logic Programs and Input Completion. A *vocabulary* is a set of propositional symbols also called atoms. As customary, a *literal* is an atom a or its negation $\neg a$. A *(propositional) logic program* over vocabulary σ is a set of *rules* of the form

$$a \leftarrow b_1, \ldots, b_\ell,\ not\ b_{\ell+1}, \ldots,\ not\ b_m,\quad not\ \ not\ b_{m+1}, \ldots,\ not\ \ not\ b_n \tag{5}$$

where a is an atom over σ or $\bot$, and each b_i, $1 \leq i \leq n$, is an atom in σ. We will sometimes use the abbreviated form for rule (5), i.e., $a \leftarrow B$, where B stands for the right hand side of the arrow in (5) and is also called a *body*. We sometimes identify body B with the propositional formula

$$b_1 \wedge \ldots \wedge b_\ell \wedge \neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n. \tag{6}$$

and rule (5) with the propositional formula $B \rightarrow a$. We call expressions $b_1 \wedge \ldots \wedge b_\ell$ and $\neg b_{\ell+1} \wedge \ldots \wedge \neg b_m$ in (6) *strictly positive* and *strictly negative* respectively. The expression a is the *head* of the rule. When a is $\bot$, we often omit it. We call such a rule *a denial*. We write $hd(\Pi)$ for the set of nonempty heads of rules in Π. We refer the reader to the paper [15] for details on the definition of an *answer set*.

We call a rule whose body is empty a *fact*. In such cases, we drop the arrow. We sometimes may identify a set X of atoms with a set of facts $\{a. \mid a \in X\}$. Also, it is customary for a given vocabulary σ, to identify a set X of atoms over σ with (i) a complete and consistent set of literals over σ constructed as $X \cup \{\neg a \mid a \in \sigma \setminus X\}$, and respectively with (ii) an assignment function or interpretation that assigns truth value *true* to every atom in X and *false* to every atom in $\sigma \setminus X$.

We now restate the definitions of input answer set and input completion [14]. These concepts are instrumental in defining constraint answer set programs and building a link towards using SMT solvers for solving such programs. In particular, it is well known that for the large class of logic programs, referred to as "tight" programs, its answer sets coincide with models of its completion, as shown by Fages [8]. A similar relation holds between input answer sets of a program and models of input completion.

▶ **Definition 1.** For a logic program Π over vocabulary σ, a set X of atoms over σ is an *input answer set* of Π relative to vocabulary $\iota \subseteq \sigma$, when X is an answer set of the program $\Pi \cup ((X \cap \iota) \setminus hd(\Pi))$.

▶ **Definition 2.** For a program Π over vocabulary σ, the *input-completion* of Π relative to vocabulary $\iota \subseteq \sigma$, denoted by $IComp(\Pi, \iota)$, is defined as the set of propositional formulas that consists of the implications $B \rightarrow a$ for all rules (5) in Π and the implications $a \rightarrow \bigvee_{a \leftarrow B \in \Pi} B$ for all atoms a occurring in $(\sigma \setminus \iota) \cup hd(\Pi)$.

Tightness is a syntactic condition on a program that can be verified by means of its dependency graph. The *dependency graph* of Π is the directed graph G such that (i) the vertices of G are the atoms occurring in Π, and (ii) for every rule $a \leftarrow B$ in Π whose head is not $\bot$, G has an edge from a to each atom in strictly positive part of B. A program is called *tight* if its dependency graph is acyclic.

▶ **Theorem 3.** *For a tight program Π over vocabulary σ and vocabulary $\iota \subseteq \sigma$, a set X of atoms from σ is an input answer set of Π relative to ι if and only if X satisfies program's input-completion $IComp(\Pi, \iota)$.*

3 EZ Constraint Answer Set Programs and Constraint Formulas

We now introduce EZ programs accepted by the CASP solver EZCSP. The EZSMT system accepts subclass of these programs (as it poses additional tightness restrictions).

Let σ_r and σ_i be two disjoint vocabularies. We refer to their elements as *regular* and *irregular* atoms respectively. For a program Π and a propositional formula F, by $At(\Pi)$ and $At(F)$ we denote the set of atoms occurring in Π and F, respectively.

▶ **Definition 4.** An *EZ constraint answer set program (or EZ program)* over vocabulary $\sigma_r \cup \sigma_i$ is a triple $\langle \Pi, \mathcal{B}, \gamma \rangle$, where

- Π is a logic program over the vocabulary $\sigma_r \cup \sigma_i$ such that
  - atoms from σ_i only appear in strictly negative part of the body[1] and
  - any rule that contains atoms from σ_i is a denial,
- $\mathcal{B}$ is a set of constraints over the same lexicon, and
- γ is an injective function from the set σ_i of irregular atoms to the set $\mathcal{B}$ of constraints.

For an EZ program $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ over $\sigma_r \cup \sigma_i$, a set $X \subseteq At(\Pi)$ is an *answer set* of P if

- X is an input answer set of Π relative to σ_i, and
- the GCSP $\{\gamma(a) | a \in X \cap \sigma_i\}$ has a solution.

This form of a definition of EZ programs is inspired by the definition of constraint answer set programs presented in [14] that generalize the CLINGCON language by Gebser et al. [11]. There are two major differences between EZCSP and CLINGCON languages. First, the later allows irregular atoms to appear in non-denials (though such atoms cannot occur in heads). Second, the third condition on answer sets of CLINGCON programs states that the GCSP $\{\gamma(a) | a \in X \cap \sigma_i\} \cup \{\neg\gamma(a) | a \in (At(\Pi) \cap \sigma_i) \setminus X\}$ has a solution.

Ferraris and Lifschitz [9] showed that a choice rule $\{a\} \leftarrow B$[2] can be seen as an abbreviation for a rule $a \leftarrow \mathit{not\ not}\ a, B$. We adopt this abbreviation.

[1] The fact that atoms from σ_i only appear in strictly negative part of the body rather than in any part of the body is inessential for the kind of constraints the EZCSP system allows.

[2] Choice rules were introduced in [19] and are commonly used in answer set programming.

Π_1	Reading of a rule
$\{switch\}.$	Action *switch* is exogenous
$lightOn \leftarrow\ switch, not\ am.$	Light is on (*lightOn*) during the night (*not am*) when *switch* has occurred.
$\leftarrow not\ lightOn.$	The light must be on.
$\{am\}.$	It is night (*not am*) or morning (*am*)
$\leftarrow not\ am,\ not\ \lvert x \geq 12\rvert.$	It must be *am* when it is not the case that $x \geq 12$ (x is understood as the hours).
$\leftarrow am,\ not\ \lvert x < 12\rvert.$	It must be *am* when it is $x < 12$.
$\leftarrow not\ \lvert x \geq 0\rvert.$	Variable x must be nonnegative.
$\leftarrow not\ \lvert x \leq 23\rvert.$	Variable x must be less than or equal to 23.

Figure 1 Program Π_1 and annotations of its rules.

▶ **Example 5.** Let us consider sample EZ program. Let Σ_2 be the numeric signature containing a single variable x. By $\mathcal{L}_2$ we denote the integer lexicon $([\Sigma_2, \mathbb{Z}], \rho_{\mathbb{Z}})$. We are now ready to define an EZ program $P_1 = \langle \Pi_1, \mathcal{B}_{\mathcal{L}_2}, \nu_1 \rangle$ over lexicon $\mathcal{L}_2$, where

- Π_1 is the program presented in Figure 1. The set of irregular atoms of Π_1 is $\{\lvert x \geq 12\rvert, \lvert x < 12\rvert, \lvert x \geq 0\rvert, \lvert x \leq 23\rvert\}$. (We use vertical bars in our examples to mark irregular atoms.) The remaining atoms form the regular set.
- $\mathcal{B}_{\mathcal{L}_2}$ is the set of all integer linear constraints over $\mathcal{L}_2$, which obviously includes constraints $\{x \geq 12, x < 12, x \geq 0, x \leq 23\}$, and
- function γ_1 is defined in intuitive manner so that for instance irregular atom $\lvert x \geq 12\rvert$ is mapped to integer linear constraint $x \geq 12$.

Consider the set

$$\{switch,\ lightOn, \lvert x \geq 12\rvert, \lvert x \geq 0\rvert, \lvert x \leq 23\rvert\} \tag{7}$$

over atoms $At(\Pi_1)$. This set is the only input answer set of Π_1 relative to its irregular atoms. Also, the integer linear constraint satisfaction problem with constraints

$$\{\gamma_1(\lvert x \geq 12\rvert), \gamma_1(\lvert x \geq 0\rvert), \gamma_1(\lvert x \leq 23\rvert)\} = \{x \geq 12, x \geq 0, x \leq 23\} \tag{8}$$

has a solution. There are 12 valuations $\nu_1 \ldots \nu_{12}$ over $\mathcal{L}_2$, which satisfy this GCSP: $x^{\nu_1} = 12$, $\ldots$, $x^{\nu_{12}} = 23$. It follows that set (7) is an answer set of P_1.

Just as we defined EZ constraints answer set programs, we can define EZ constraint formulas.

▶ **Definition 6.** An *EZ constraint formula* over the vocabulary $\sigma_r \cup \sigma_i$ is a triple $\langle F, \mathcal{B}, \gamma \rangle$, where

- F is a propositional formula over the vocabulary $\sigma_r \cup \sigma_i$,
- $\mathcal{B}$ is a set of constraints over the same lexicon, and
- γ is an injective function from the set σ_i of irregular atoms to the set $\mathcal{B}$ of constraints.

For a constraint formula $\mathcal{F} = \langle F, \mathcal{B}, \gamma \rangle$ over $\sigma_r \cup \sigma_i$, a set $X \subseteq At(F)$ is a *model* of $\mathcal{F}$ if

- X is a model of F, and
- the GCSP $\{\gamma(a) | a \in X \cap \sigma_i\}$ has a solution.

Following theorem captures a relation between EZ programs and EZ constraint formulas. This theorem is an immediate consequence of Theorem 3.

▶ **Theorem 7.** *For an EZ program $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ over the vocabulary $\sigma = \sigma_r \cup \sigma_i$ and a set X of atoms over σ, when Π is tight, X is an answer set of P if and only if X is a model of EZ constraint formula $\langle IComp(\Pi, \sigma_i), \mathcal{B}, \gamma \rangle$ over σ.*

In the sequel, we will abuse the term "tight". We will refer to an EZ program $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ as *tight* when its first member Π has this property.

Linear and Integer Linear EZ Programs. We now review the more refined details behind programs supported by EZCSP. These EZ programs are of particular form:

1. $\langle \Pi, \mathcal{B}_{\mathcal{L}}, \gamma \rangle$, where $\mathcal{L}$ is a numeric lexicon and $\mathcal{B}_{\mathcal{L}}$ is the set of all linear constraints over $\mathcal{L}$, or
2. $\langle \Pi, \mathcal{B}_{\mathcal{L}}, \gamma \rangle$, where $\mathcal{L}$ is an integer lexicon and $\mathcal{B}_{\mathcal{L}}$ is the set of all integer linear constraints over $\mathcal{L}$.

We refer to the former as *EZ programs modulo linear constraints (or EZ(L) programs)*, whereas to the latter as *EZ programs modulo integer linear constraints (or EZ(IL) programs)*. Similarly, we can define EZ constraint formulas modulo linear constraints and EZ constraint formulas modulo integer linear constraints. Lierler and Susman [14] showed that such constraint formulas coincide with formulas in satisfiability modulo linear arithmetic, or SMT(L), and satisfiability modulo integer linear arithmetic, or SMT(IL), respectively.

The EZ program P_1 from Example 5 is an EZ(IL) program. Listing 1 presents this program in the syntax accepted by the EZCSP solver. We refer to this syntax as the EZCSP language. Line 1 in Listing 1 specifies that this is an EZ(IL) program. Line 2, first, declares that variable x is in the signature of program's integer lexicon. Second, it specifies that x may be assigned values in range from 0 to 23. Thus, Line 2 essentially encodes the last two rules in Π_1 presented in Figure 1. Lines 3-6 follow the first four lines of Π_1 modulo replacement of symbol $\leftarrow$ with symbols `:-`. In the EZCSP language, all irregular atoms are enclosed in a "required" statement and are syntactically placed in the head of their rules. So that Lines 7 and 8 encode the last two rules of Π_1, respectively. If a denial of an EZ program contains more than one irregular atom then in the EZCSP language disjunction in required statement is used to encode such rules. For instance, an EZ rule

$$\leftarrow \ not\ |x > 5|,\ not\ |x < 12|$$

has the form $required(x > 5 \vee x < 12)$. in the EZCSP syntax. (One may also use conjunction and implication within the required syntax.)

```
1 cspdomain(fd).
2 cspvar(x,0,23).
3 {switch}.
4 lightOn :- switch not am.
5 :- not lightOn.
6 {am}.
7 required(x ≥ 12) :- not am.
8 required(x < 12) :- am.
```

Listing 1 EZCSP Program.

4 The EZSMT Solver

By Theorem 7, it follows that answer sets of a tight EZ program coincide with models of a constraint formula that corresponds to the input completion of the EZ program relative to its irregular atoms. Thus, tight EZ(L) and EZ(IL) programs can be converted to "equivalent" SMT(L) and SMT(IL) formulas, respectively. This fact paves a way to utilizing SMT technology for solving tight EZ programs. The EZSMT system introduced in this work roots on these ideas.

In a nutshell, the EZSMT system takes a tight EZ(L) or EZ(IL) program written in the EZCSP language and produces an equivalent SMT(L) or SMT(IL) formula written in the SMT-LIB language that is a common input language for SMT solvers [4]. Subsequently,

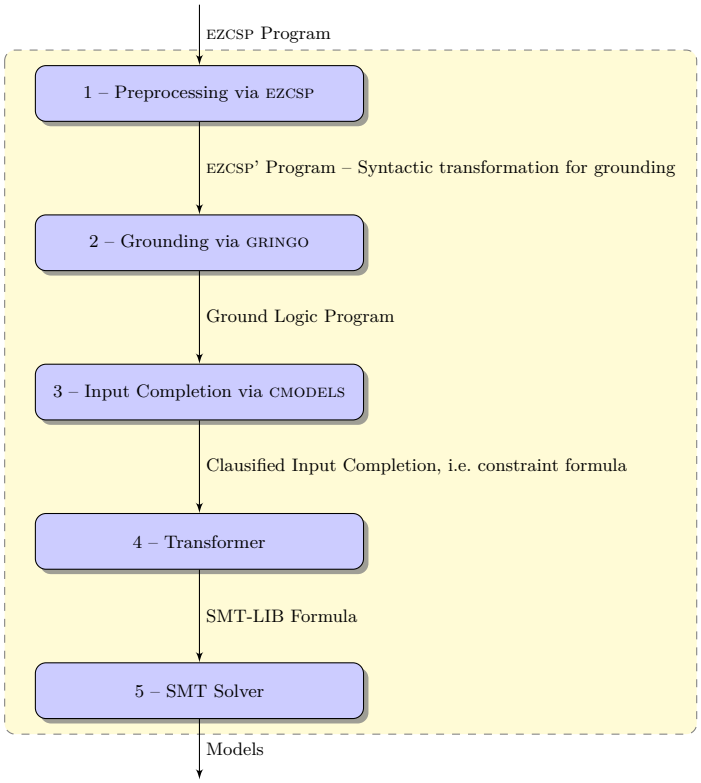

Figure 2 EZSMT Pipeline.

EZSMT runs a compatible SMT solver, such as CVC4 [3] or Z3 [6], to compute models of the program.

Few remarks are due with respect to the SMT-LIB language. This language allows the SMT research community to develop benchmarks and run solving competitions using standard interface of common input language. Barret et al. [4] define the syntax and usage of SMT-LIB. As opposed to constraint answer set programming languages, which are regarded as declarative *programming* languages, SMT-LIB is a low-level specification language. It is not intended to be a modeling language, but geared to be easily interpretable by SMT solvers and serve as a standard interface to these systems. As such, this work provides an alternative to SMT-LIB for utilizing SMT technology. It advocates the use of tight EZ programs as a declarative programming interface for SMT solvers. Also the availability of SMT-LIB immediately enables its users to interface multiple SMT-solvers as off-the-shelve tools without the need to utilize their specialized APIs.

The EZSMT Architecture. We now present details behind the EZSMT system. Figure 2 illustrates its pipeline. We use the EZ program from Example 5 to present a sample workflow of EZSMT.

Preprocessing and Grounding. In this paper, we formally introduced EZ programs over a signature that allows propositional atoms or irregular atoms. In practice, EZCSP language, just as traditional answer set programming languages, allows the users to utilize non-irregular atoms with schematic-variables. The process of eliminating these variables is referred to as *grounding* [10]. It is a well understood process in answer set programming and off the

shelf grounders exist, e.g., the GRINGO system[3] [10]. The EZSMT solver also allows schematic-variables (as they are part of the EZCSP language) and relies on GRINGO to eliminate these variables.

Prior to applying GRINGO, all irregular atoms in the input program must be identified to be properly processed while grounding. The "required" keyword in the EZCSP language allows us to achieve this so that the rules with the "required" expression in the head are converted into an intermediate language. The invocation of the EZCSP system with the `--preparse-only` flag performs the conversion. The preprocessing performed by EZCSP results in a valid input program for the grounder GRINGO.

For instance, the application of EZCSP with `--preparse-only` flag on the program in Listing 1 results in the program that replaces last two rules of original program by the following rules

```
required(ezcsp__geq(x, 12)) :- not am.
required(ezcsp__lt(x, 12)) :- am.
```

Program's Completion. The third block in the pipeline in Figure 2 is responsible for three tasks. First, it determines whether the program is tight or not. Given a non tight program the system will exit with the respective message. Second, it computes the input completion of a given program (recall, that this input completion can be seen as an SMT program). Third, the input completion is clausified using Tseitin transformations so that the resulting formula is in conjunctive normal form. This transformation preserves the models of the completion modulo original vocabulary. The output from this step is a file in a DIMACS[4]-inspired format. System CMODELS [12] is used to perform the described steps. It is invoked with the `--cdimacs` flag.

For example, given the grounding produced by GRINGO for the preprocessed program in Listing 1, CMODELS will produce the output presented in Listing 2. This output encodes the clausified input completion of the EZ program in Example 5 and can be viewed as an SMT formula.

```
smt cnf 5 8
-switch switch 0
-switch lightOn 0
-lightOn switch 0
cspdomain(fd) 0
cspvar(x,0,23) 0
switch 0
lightOn 0
required(ezcsp__geq(x,12)) 0
```

Listing 2 Completion of EZCSP Program.

The first line in Listing 2 states that there are 5 atoms and 8 clauses in the formula. Each other line stands for a clause, for instance, line `-switch switch 0` represents clause $\neg switch \vee switch$.

It is important to note that just as the EZCSP language accepts programs with schematic variables, it also accepts programs with so called weight and cardinality constraint rules introduced in [19]. System CMODELS eliminates such rules in favor of rules of the form (5)

[3] `http://potassco.sourceforge.net`
[4] `http://www.satcompetition.org/2009/format-benchmarks2009.html`

discussed here. (The translation used by CMODELS was introduced in [9].) Thus, solver EZSMT is capable of accepting programs that contain weight and cardinality constraint rules.

Transformation. The output program from CMODELS serves as input to the Transformer block in the EZSMT pipeline. Transformer converts the SMT formula computed by CMODELS into the SMT-LIB syntax. For instance, given the SMT program presented in Listing 2, the Transformer produces the following SMT-LIB code.

```
1
2   (set-option :interactive-mode true)
3   (set-option :produce-models true)
4   (set-option :produce-assignments true)
5   (set-option :print-success false)
6   (check-sat)
7   (get-model)
8   (set-logic QF_LIA)
9   (declare-fun |lightOn| () Bool)
10  (declare-fun |required(ezcsp__geq(x,12))| () Bool)
11  (declare-fun |switch| () Bool)
12  (declare-fun |cspvar(x,0,23)| () Bool)
13  (assert (or (not |switch|) |switch|))
14  (assert (or (not |switch|) |lightOn|))
15  (assert (or (not |lightOn|) |switch|))
16  (assert |cspvar(x,0,23)|)
17  (assert |switch|)
18  (assert |lightOn|)
19  (assert |required(ezcsp__geq(x,12))|)
20  (declare-fun |x| () Int)
21  (assert (=> |required(ezcsp__geq(x,12))| (>= |x|  12)))
22  (assert (=> |cspvar(x,0,23)| (<= 0 |x|)))
23  (assert (=> |cspvar(x,0,23)| (>= 23 |x|)))
```

The resultant SMT-LIB specification can be described as follows:

(i) Lines 1–6 are responsible for setting directives necessary to indicate to an SMT solver that it should find a model of the program after satisfiability is determined [4].

(ii) In line 7, the Transformer instructs an SMT solver to use quantifier-free linear integer arithmetic (*QF_LIA*) [4] to solve given SMT(IL) formula. (The clause `cspdomain(fd) 0` from Listing 2 serves as an indicator that the given formula is an SMT(IL) formula.)

(iii) Lines 8–11 are declarations of the atoms in our sample program as boolean variables (called functions in the SMT-LIB parlance).

(iv) Lines 12–18 assert the clauses from Listing 2 to be true.

(v) Line 19 declares variable x to be an integer.

(vi) Line 20 expresses the fact that if the irregular atom `required(ezcsp__geq(x,12))` holds then the constraint $x \geq 12$ must also hold. In other words, it plays a role of a mapping γ_1 from Example 5.

(vii) Lines 21–22 declare the domain of variable x to be in range from 0 to 23 (recall how Listing 1 encodes this information with `cspvar(x,0,23)`).

SMT Solver. The final step is to use an SMT solver that accepts input in SMT-LIB. The output produced by CVC4[5] given the SMT-LIB program listed last follows:

[5] We note that the output format of the SMT solver Z3 is of the same style as that of CVC4.

```
sat
(model
(define-fun lightOn () Bool true)
(define-fun |required(ezcsp__geq(x,12))| () Bool true)
(define-fun switch () Bool true)
(define-fun |cspvar(x,0,23)| () Bool true)
(define-fun x () Int 12))
```

The first line of the output indicates that a satisfying assignment exists. The subsequent lines present a model that satisfies the given SMT-LIB program. Note how this model corresponds to answer set (7). Also, the solver identified one of the possible valuations for x that satisfies integer linear constraint satisfaction problem (8), this valuation maps x to 12.

Limitations. Due to the fact that the EZSMT solver accepts programs in the EZCSP language, it is natural to compare the system to the EZCSP solver. The EZSMT system faces some limitations relative to EZCSP. The EZSMT solver accepts only a subset of the EZCSP language. In particular, it supports a limited set of its global constraints [2]. Only, the global constraints *all_different* and *sum* are supported by EZSMT. Also, EZSMT can only be used on tight EZCSP programs. Yet, we note that this is a large class of programs. No support for minimize and maximize statements of EZCSP or GRINGO languages is present. In addition, solver EZSMT computes only a single answer set. Modern SMT solvers are often used for establishing satisfiability of a given formula rather than for finding its models. For instance, the SMT-LIB language does not provide a directive to instruct an SMT solver to find all models for its input. To bypass this obstacle one has to promote (i) the extensions of the SMT-LIB standard to allow a directive for computing multiple models as well as (ii) the support of this functionality by SMT solvers. Alternatively, one may abandon the use of SMT-LIB and utilize the specialized APIs of SMT solvers in interfacing these systems. The later solution seems to lack the generality as it immediately binds one to peculiarities of APIs of distinct software systems. Addressing mentioned limitations of EZSMT is a direction of future work.

5 Experimental Results

In order to demonstrate the efficacy of the EZSMT system and to provide a comparison to other existing CASP solvers, six problems have been used to benchmark EZSMT. The first three benchmarks stem from the Third Answer Set Programming Competition, 2011[6] (ASPCOMP). The selected encodings are: *weighted sequence, incremental scheduling,* and *reverse folding*. Balduccini and Lierler [2] use these three problems to assess performance of various configurations of the EZCSP and CLINGCON systems. We utilize the encodings for EZCSP and CLINGCON stemming from this earlier work for these problems. We also adopted these encodings to fit the syntax of the MINGO language to experiment with this system. The last three benchmarks originate from the assessment of solver MINGO [16]. This system translates CASP programs into mixed integer programming formalism and utilizes IBM ILOG CPLEX[7] system to find solutions. The selected problems are: *job shop, newspaper,* and *sorting*. We used the encodings provided in [16] for MINGO, CLINGCON, and CMODELS. We adopted the CLINGCON encoding to fit the syntax of the EZCSP language to experiment with EZCSP and EZSMT. All six mentioned benchmarks do not scale when using traditional answer

[6] https://www.mat.unical.it/aspcomp2011
[7] http://www.ibm.com/software/commerce/optimization/cplex-optimizer/

Table 2 ASPCOMP 2011 and MINGO Benchmarks

Benchmark (number of instances)	EZSMT CVC4	EZSMT Z3	CLINGCON	EZCSP	MINGO	CMODELS
	Cumulative Time (timeout)					
Reverse folding (50)	47948 (22)	4873 (2)	2014 (1)	**559**	14962 (1)	84616 (47)
Weighted Seq. (30)	24.2	**23.3**	187	13879	1330	54000 (30)
Incr. scheduling (30)	10277 (5)	**9135 (5)**	20417 (11)	37332 (20)	13626 (7)	54000 (30)
Job shop (100)	106	48.8	**2.77**	180000 (100)	1137	163106 (90)
Newspaper (100)	7.68	3.77	**0.02**	3.53	54.2	111615 (53)
Sorting (189)	646	233	**31.7**	103	8282	271004 (141)

set solvers. The EZSMT system, encodings, and instances used for benchmarking are available at the EZSMT site: `http://unomaha.edu/nlpkr/software/ezsmt/`.

All experiments were conducted on a computer with an Intel Core i7-940 processor running Ubuntu 14.04 LTS (64-bit) operating system. Each benchmark was allocated 4 GB RAM, a single processor core, and given an 1,800 second timeout. No benchmarks were run simultaneously.

Five CASP solvers and one answer set (ASP) solver were benchmarked:

- EZSMT v. 1.0 with CVC4 v. 1.4 as the SMT solver (EZSMT– CVC4),
- EZSMT v. 1.0 with Z3 v. 4.4.2 – 64 bit as the SMT solver (EZSMT– Z3),
- CLINGCON v. 2.0.3 with constraint solver GECODE v. 3.7.3 and ASP solver CLASP v. 1.3.10,
- EZCSP v. 1.6.20 with constraint solver B-Prolog v. 7.4 #3 and ASP solver CMODELS v. 3.86,
- MINGO v. 2012-09-30 with mixed integer solver CPLEX v. 12.5.1.0, and
- ASP solver CMODELS v. 3.86 [12].

All of these systems invoke grounder GRINGO versions 3.0.+ during their executions. Time spent in grounding is reported as part of the solving time. The best performing EZCSP configuration, as reported in [2], was used for each run of the ASPCOMP benchmarks. All other systems were run under their default configurations. We note that for systems EZSMT-CVC4, EZSMT-Z3, and EZCSP identical encodings across the benchmarks were used. The formalizations for other solvers can be seen as syntactically different versions of these encodings.

At a high-level abstraction, one may summarize the architectures of the CLINGCON and EZCSP solvers as *ASP-based solvers plus constraint solver*. Given a constraint answer set program $\langle \Pi, \mathcal{B}, \gamma \rangle$, both CLINGCON and EZCSP first use an answer set solver to (partially) compute an input answer set of Π. Second, they contact a constraint solver to verify whether respective GCSP has a solution. As mentioned earlier, MINGO's solving is based on mixed integer programming.

Table 2 presents the experimental results. Each name of a benchmark is annotated with the number of instances used in the experiments. The collected running times are reported in cumulative fashion. The number in parenthesis annotates the number of timeouts or memory outs (that we do not distinguish). Any instance which timed-out/memory-out is represented in cumulative time by adding the maximum allowed time for an instance (1,800 seconds). For instance, answer set solver CMODELS timed out on all 30 instances of the weighted sequence

benchmark so that the cumulative time of 54,000 is reported. Bold font is used to mark the best performing solver.

In the *reverse folding* benchmark, the difference between SMT solvers used for EZSMT becomes very apparent. In this case, the Z3 solver performed better than CVC4 by an order of magnitude. This underlines both the importance of solver selection and difference between SMT solvers. These observations mark the significance of the flexibility that EZSMT provides to its users as they are free to select different SMT solvers as appropriate to the instances and encodings. Indeed, SMT solvers are interfaced via the standard SMT-LIB language by EZSMT.

In the *weighted sequence* benchmark, we note that no CASP system timed out. In this case, the EZSMT system features a considerable speedup. It noticeably outperforms CLINGCON and EZCSP by multiple orders of magnitude.

In *incremental scheduling*, the original EZCSP encoding includes a global constraint, cumulative, which is not supported by EZSMT. To benchmark EZSMT on this problem, the encoding was rewritten to mimic a method used in the CLINGCON encoding that also does not support the cumulative global constraint. Columns EZSMT-CVC4, EZSMT-Z3, EZCSP in Table 2 represent instances run on the rewritten encoding. Solver EZSMT times out the least, followed by CLINGCON timing out on over one-third the instances, and finally EZCSP, which times out on about half the instances. We note that on the original encoding with cumulative constraint EZCSP performance is captured by the following numbers 26691 (14). Thus, the use of the cumulative global constraint allowed EZCSP to run more instances to completion. All solvers time out on the same 5 instances, which EZSMT-CVC4 and EZSMT-Z3 times out on.

The last three lines in Table 2 report on the three benchmarks from [16]. In general, we observe that CLINGCON features the fastest performance, followed by EZSMT and MINGO for these benchmarks.

Overall, the benchmarks reveal several aspects of the EZSMT solver. First, as demonstrated by the reverse folding results in Table 2, the underlying SMT solving technology selected for the SMT-LIB program produced by EZSMT is important. Next, the weighted sequence and the incremental scheduling results demonstrate the efficacy of EZSMT approach. Furthermore, Table 2 shows that EZSMT outperforms MINGO across the board.

The ASPMT2SMT system [5] is closely related to EZSMT in a sense that it utilizes SMT technology for finding solutions to first order formulas under stable model semantics forming so called ASPMT language. The EZ programs can be seen as a special case of ASPMT formulas. Just as EZSMT poses restriction on its programs to be tight, ASPMT2SMT poses a similar restriction on its theories. The ASPMT2SMT solver utilizes SMT solver Z3 to find models of ASPMT theories by interfacing this system via its API. This tight integration with Z3 allows ASPMT2SMT to find multiple/all models of its theories in contrast to EZSMT. Yet, the fact that EZSMT advocates the use of the standard SMT-LIB language makes its approach more open towards new developments in the SMT solving technology as it is not tied to any particular SMT solver via its specific API. We do not present the times for the ASPMT2SMT system as the ASPMT language differs from the input languages of other systems that we experimented with so that encodings of our benchmarks for ASPMT2SMT are not readily available. Yet, EZSMT-Z3 times should mimic these by ASPMT2SMT as both systems rely on forming program's completion in the process of translating inputs in their respective languages into SMT formulas. Verifying this claim is part of the future work.

6 Conclusions and Future Work

This work presents the EZSMT system, which is able to take tight constraint answer set programs and rewrite them into the SMT-LIB formulas that can be then processed by SMT solvers. The EZSMT solver parallels the efforts of the ASPMT2SMT system [5] that utilizes SMT technology for solving programs in related formalism. Our experimental analysis illustrates that the EZSMT system is capable of outperforming other cutting-edge CASP solvers. Niemela [18] characterized answer sets of "normal" logic programs in terms of "level rankings" and developed a mapping from such programs to so called difference logic. Mapping of the kind has been previously exploited in the design of solvers DINGO [13] and MINGO [16]. We believe that these ideas are applicable in the settings of EZ(IL) and EZ(L) programs. Verifying this claim and adopting the results within EZSMT to allow this solver to process non tight programs is the direction of future work.

Acknowledgments. We would like to thank Martin Brain for the discussions that led us to undertake this research.

References

1 Marcello Balduccini. Representing constraint satisfaction problems in answer set programming. In *ICLP Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP)*, 2009. URL: `https://www.mat.unical.it/ASPOCP09/`.

2 Marcello Balduccini and Yuliya Lierler. Constraint answer set solver EZCSP and why integration schemas matter. Unpublished draft, available at `https://works.bepress.com/yuliya_lierler/64/`, 2016.

3 Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), volume 6806 of LNCS*. Springer, 2011.

4 Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015.

5 Michael Bartholomew and Joohyung Lee. System aspmt2smt: Computing aspmt theories by smt solvers. In *European Conference on Logics in Artificial Intelligence, JELIA*, pages 529–542. Springer, 2014. `doi:10.1007/978-3-319-11558-0_37`.

6 Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

7 Christian Drescher and Toby Walsh. A translational approach to constraint answer set solving. *Theory and Practice of Logic programming (TPLP)*, 10(4-6):465–480, 2010.

8 François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

9 Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.

10 Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in gringo series 3. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 345–351. Springer, 2011. `doi:10.1007/978-3-642-20895-9_39`.

11 Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint answer set solving. In *Proceedings of 25th International Conference on Logic Programming*, pages 235–249. Springer, 2009.

12 Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36:345–377, 2006.

13 Tomi Janhunen, Guohua Liu, and Ilkka Niemela. Tight integration of non-ground answer set programming and satisfiability modulo theories. In *Proceedings of the 1st Workshop on Grounding and Transformations for Theories with Variables*, 2011.

14 Yuliya Lierler and Benjamin Susman. Constraint answer set programming versus satisfiability modulo theories. In *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, 2016.

15 Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

16 Guohua Liu, Tomi Janhunen, and Ilkka Niemela. Answer set programming via mixed integer programming. In *Knowledge Representation and Reasoning Conference*, 2012. URL: `https://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4516`.

17 Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

18 Ilkka Niemelä. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence*, 53:313–329, 2008.

19 Ilkka Niemelä and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.